CSI 466 - Teoria dos Grafos
Busca em Largura e em Profundidade

Lucas Monteiro Martino Cota[1]

[1]Instituto de Ciências Exatas e Aplicadas (ICEA)
Universidade Federal de Ouro Preto (UFOP) - João Monlevade – MG – Brasil

lucas.cota@aluno.ufop.edu.br

Abstract. *Searching for a better solution in a set of possible solutions is one of the pillars of Artificial Intelligence (AI), this principle can be represented through graphs using search methods, such as breadth and in depth search, to find the best solution. Thus, it's possible to apply this idea in several logical reasoning games such as chess, tic-tac-toe and maze. In this article, two graphs will be used to compare the search techniques and what is the best representation, either by list or matrix of adjacencys. The results obtained in the experiments performed will be analyzed in terms of execution time, amount of memory used and among other aspects.*

Resumo. *Buscar uma melhor solução em um conjunto de soluções possíveis é um dos pilares da Inteligência Artificial (IA), este princípio pode ser representado através de grafos utilizando métodos de busca, como busca em largura e em profundidade, para achar a melhor solução. Sendo assim, é possível aplicar esta ideia em diversos jogos de raciocínio lógico como xadrex, jogo da velha e labirinto. Neste artigo dois grafos serão utilizados para comparar as técnicas de busca e qual a melhor representação, seja por lista ou matriz de adjacências. Os resultados obtidos através dos experimentos realizados vão ser analisados em termos do tempo de execução, quantidade de memória gasta e entre outros aspectos.*

1. Introdução

A Teoria do Grafos é uma disciplina que aborda o estudo das relações entre objetos de um determinado conjunto através de grafos G(V, E, w). Ao começar o estudo de grafos você percebe que vários problemas da vida real podem ser modelados em grafos, tornando mais fácil o seu entendimento e a sua resolução.

Do mesmo modo que é simples transformar um problema real em um grafo, também é simples representar este grafo computacionalmente e se, um problema real em forma de grafo já era mais fácil de se lidar, com a ajudar de um computador fica mais ainda. Hoje em dia existem diversos algoritmos que ajudam na solução dos mais variados desafios que podem vir a aparecer, mas também é verdade que ainda existem aqueles problemas que não foram solucionados.

Os algoritmos de busca estão entre os que auxiliam na resolução dos mais variados problemas do dia a dia, podendo ser responsável por achar uma melhor solução, como em jogos ou então se juntando com outros algoritmos para destrinchar desafios

mais complexos. Neste artigo será dado ênfase justamente aos algoritmos de busca, mais especificamente os de Busca em Largura e Busca em Profundidade.

O objetivo do artigo é comparar e analisar estes métodos de busca e suas representações computacionais utilizando dois casos da vida real que foram modelados em grafos. Os grafos utilizados foram um grafo de colaborações em pesquisa, que é composto por um vértice para cada pesquisador e arestas caso tenham publicado artigos científicos juntos e um grafo de conexões da web, que contém as conexões das redes que formam a Internet.

Ao final dos experimentos será possível perceber as vantagens e desvantagens dos tipos de busca abordados, como também qual a melhor forma de representar grafos computacionalmente, tornando mais fácil a escolha de qual tipo de busca utilizar e qual forma de representação do grafo dependendo do problema que tem de ser resolvido.

2. Algoritmos

Nesta seção as funções do código utilizado no artigo vão ser detalhadas e explicadas para um melhor entendimento do programa e de como ele deve ser utilizado. Como as operações realizadas sobre lista e matriz são muito parecidas, apenas as funções que realizam modificações em listas vão ser apresentadas, o código fonte completo e os arquivos com os grafos utilizados para os experimentos do artigo podem ser visualizados em um repositório no GitHub[COTA 2021].

A primeira parte do código exibida abaixo é o início da main, onde nas linhas (1-7) o programa pede para que o usuário entre com o nome do arquivo a ser lido, logo depois a variável **cabeçalho** lê a primeira linha deste arquivo, que tem quer ser o número de vértices e de arestas do grafo e separa estes números em duas variáveis diferentes. Se tudo tiver ocorrido corretamente, nas linhas (9-18) o programa pede para que o usuário escolha o modo de representação do grafo até que ele digite uma opção válida.

Algoritmo 1. Início da main

```
1  nomeArquivo = input("Digite o nome do arquivo que voce deseja ler: ")
2  arquivo = open(nomeArquivo, 'r')
3
4  cabecalho = arquivo.readline()
5  cabecalho = cabecalho.split()
6  numVertices = int(cabecalho[0])
7  numArestas = int(cabecalho[1])
8
9  print("Escolha a maneira que voce deseja representar o seu grafo: ")
10 print("1 - Lista de Adjacencias")
11 print("2 - Matriz de Adjacencias")
12 print("Digite sua opcao: ", end="")
13 opcao = int(input())
14
15 while(opcao != 1 and opcao != 2):
16     print("Opcao invalida!")
17     print("Digite novamente: ", end="")
18     opcao = int(input())
```

Após a escolha da forma de representação do grafo, nas linhas (1-16) estão as operações que serão realizadas sobre a lista de adjacência e nas linhas (17-32) sobre a matriz, dependendo da escolha do usuário e na linha (34) o arquivo é fechado, encerrando o programa.

Note que as variáveis **lista** e **matriz** vão funciunar como variáveis auxiliares e o grafo vai estar corretamente armazenado nas variáveis **listaAdjacencia** e **matAdj**. O vetor global de vértices descobertos **desc** é inicializado do tamanho do grafo e com 0 em todas as posições, e antes de ser realizada qualquer busca todas as posições são zeradas na função **zerarDescobertos()**.

Algoritmo 2. main

```
1  if opcao == 1:
2      lista = []
3      listaAdjacencia = [[] for i in range(numVertices)]
4      lista.append(arquivo.readlines())
5      preencheListaAdjacencia(lista, listaAdjacencia)
6      desc = [0 for i in range(len(listaAdjacencia))]
7      input("\nAperte enter para exibir as informacoes do grafo !\n")
8      exibirInformacoes(listaAdjacencia)
9      v = int(input("\nDigite um vertice para comecar a buscar em largura
           : "))
10     busca_largura(listaAdjacencia, v)
11     v = int(input("\nDigite um vertice para comecar a buscar em
           profundidade: "))
12     zerarDescobertos()
13     busca_profundidade(listaAdjacencia, v)
14     input("\nAperte enter para exibir as informacoes dos grafos conexos
           !\n")
15     zerarDescobertos()
16     informacoesGrafosConexos(listaAdjacencia)
17 else:
18     matriz = []
19     matAdj = [[0 for i in range(numVertices)]for i in range(numVertices
           )]
20     matriz.append(arquivo.readlines())
21     matAdj = preencheMatrizAdjacencia(matriz, matAdj)
22     desc = [0 for i in range(len(matAdj))]
23     input("\nAperte enter para exibir as informacoes do grafo !\n")
24     exibirInformacoes(matAdj)
25     v = int(input("\nDigite um vertice para comecar a buscar em largura
           : "))
26     busca_larguraMA(matAdj, v)
27     v = int(input("\nDigite um vertice para comecar a buscar em
           profundidade: "))
28     zerarDescobertos()
29     busca_profundidadeMA(matAdj, v)
30     input("\nAperte enter para exibir as informacoes dos grafos conexos
           !\n")
31     zerarDescobertos()
32     informacoesGrafosConexosMA(matAdj)
33
34 arquivo.close()
```

As funções para preencher a lista ou matriz funcionam de forma parecida, ambas vão iterando sobre cada elemento da lista em que o arquivo foi lido, onde cada iteração corresponde a uma linha do arquivo. A função **split()** é utilizada para ajudar na separação da linha em dois vértices e um peso entre eles, após a separação os vértices são adicionados ao seu devido lugar. Como em uma matriz as alterações não ficam salvas, é preciso retornar a matriz preenchida, linha (27).

Algoritmo 3. Preencher lista

```
1  def preencheListaAdjacencia(lista, listaAdjacencia):
2
3      for i in lista[0]:
4          separador = i.split()
5
6          v1 = (int(separador[0]))
7          v2 = (int(separador[1]))
8          peso = (int(separador[2]))
9
10         if (v2, peso) not in listaAdjacencia[v1]:
11             listaAdjacencia[v1].append((v2, peso))
12         if (v1, peso) not in listaAdjacencia[v2]:
13             listaAdjacencia[v2].append((v1, peso))
```

Para exibir as informações requisitadas do grafo também foi criada uma função. Nas linhas (5-10) o grau de cada vértice é contado e armazenado na lista **listaGraus**, logo depois o maior e menor grau são exibidos, assim como o grau médio. Em caso de vértices com menor ou maior grau iguais apenas o primeiro da lista será exibido.

Nas linhas (18-21) os graus encontrados são armazenados na lista **grausUnicos**, mas esta lista vai conter apenas uma ocorrência de cada grau, ela também é ordenada para os graus aparecerem em ordem crescente. As linhas (23-30) preenchem uma lista **x** de graus possíveis e uma lista **freqRel** da frequência relativa de cada um desses graus e essas duas listas são usadas para exibir um gráfico com a distribuição empírica do grafo nas linhas (32-36). Já nas linhas (40-41) a frequência relativa de cada grau é imprimida, contando a quantidade de vezes que o grau apareceu na lista **listaGraus** e dividindo esta quantidade pelo tamanho da lista.

Algoritmo 4. Exibindo informações

```
1  def exibirInformacoes(grafo):
2      listaGraus = []
3      grausUnicos = []
4
5      for i in grafo:
6          grau = 0
7          for j in i:
8              if j != 0:
9                  grau += 1
10         listaGraus.append(grau)
11
12     print("Maior grau:", max(listaGraus), "- vertice: ",
13           listaGraus.index(max(listaGraus)))
14     print("Menor grau:", min(listaGraus), "- vertice: ",
15           listaGraus.index(min(listaGraus)))
```

```
16      print("\nGrau medio:", sum(listaGraus) / len(listaGraus))
17
18      for i in listaGraus:
19          if i not in grausUnicos:
20              grausUnicos.append(i)
21      grausUnicos.sort()
22
23      freqRel = []
24      x = []
25      for i in range(len(listaGraus)):
26          x.append(i)
27          freqRel.append(0)
28
29      for i in grausUnicos:
30          freqRel[i] = (listaGraus.count(i) / len(listaGraus))
31
32      plt.plot(x, freqRel)
33      plt.title('Distribuicao empirica')
34      plt.xlabel('Grau')
35      plt.ylabel('Frequencia relativa')
36      plt.show()
37
38      input("\nAperte enter para exibir a frequencia relativa de cada
            grau !\n")
39      print("\nFrequencia relativa:")
40      for i in grausUnicos:
41          print("Grau", i, ":", listaGraus.count(i) / len(listaGraus))
```

A próxima função é o algoritmo de busca em largura, ele recebe o grafo que será realizada a busca e o vértice origem **s** em que a busca inicia. Além de realizar a pesquisa no grafo, as funções de busca também vão ser responsáveis por escrever em um arquivo de saída os vértices descobertos e o seu respectivo nível.

Nas linhas (2-8) ocorrem as inicializações das variáveis necessárias para o algoritmo funcionar corretamente. Das linhas (9-17) ocorre o processo de busca em largura, onde a iteração o primeiro elemento de **Q** é removido e armazenado em **u**, cada vértice adjacente a **u** que ainda não foi descoberto é adicionado a **Q** e a **R** que é o vetor de vértices descobertos.

Quando **Q** estiver vazio a busca é encerrada e nas linhas (18-20) são escritos os vértices descobertos com seus níveis no arquivo de saída. Na lista **string** cada elemento é uma linha a ser escrita e a lista **nivel** é utilizada para armazenar corretamente o nível de cada vértice. Na linha (21) o arquivo é fechado.

Algoritmo 5. Busca em largura

```
1  def busca_largura(G, s):
2      arquivo = open('busca_largura.txt', 'w')
3      string = [[] for i in range((len(G)))]
4      nivel = [0 for i in range(len(G))]
5      Q = [s]
6      R = [s]
7      desc[s] = 1
8      string[0] = "vertice: nivel\n" + str(s) + ": " + str(nivel[s]) + "\
           n"
```

```
9       while len(Q) != 0:
10          u = Q.pop(0)
11          for v, i in G[u]:
12              if desc[v] == 0:
13                  nivel[v] = nivel[u] + 1
14                  string[v] = str(v) + ": " + str(nivel[v]) + "\n"
15                  Q.append(v)
16                  R.append(v)
17                  desc[v] = 1
18      for i in string:
19          if len(i) > 0:
20              arquivo.write(i)
21      print("\nMaior nivel: ", max(nivel))
22      arquivo.close()
23      return R
```

O algoritmo de busca em profundidade funciona de forma semelhante ao de busca em largura, mas ao invés de ir nos vértices adjacentes, ela vai percorrendo até o vértice de nível mais alto.

Nas linhas (2-8) ocorrem as inicializações e nas linhas (9-23) acontece a busca. A variável **u** vai receber o último elemento de **S** e vai explorando até que chegue ao vértice de nível mais alto, neste momento a variável **desempilhar** vai ter o valor true, o que faz com que ocorra o desempilhamento do vetor **S** e a busca se encerra quando **S** estiver vazio..

Nas linhas (24-26) acontece a escrita no arquivo de saída, com a lista **string** funcionando da mesma maneira que na função de busca em largura e a variável **cont** sendo utilizada para armazenar os níveis dos vértices, de modo análogo a lista **nivel** na função de busca em largura.

Algoritmo 6. Busca em profundidade

```
1   def busca_profundidade(G, s):
2       arquivo = open('busca_profundidade.txt', 'w')
3       string = [[] for i in range((len(G)))]
4       S = [s]
5       R = [s]
6       desc[s] = 1
7       string[0] = "vertice: nivel\n" + str(s) + ": 0\n"
8       cont = 1
9       while len(S) != 0:
10          u = S[-1]
11          desempilhar = True
12          for v, i in G[u]:
13              if desc[v] == 0:
14                  string[v] = str(v) + ": " + str(cont) + "\n"
15                  desempilhar = False
16                  S.append(v)
17                  R.append(v)
18                  desc[v] = 1
19                  cont += 1
20                  break
21          if desempilhar:
22              S.pop()
```

```
            cont -= 1
    for i in string:
        if len(i) > 0:
            arquivo.write(i)
    arquivo.close()
    return R
```

As próximas duas funções são utilizadas para descobrir as componentes conexas do grafo. A função **componentes-conexas** utiliza a busca em largura para isso, a cada componente conexa a variável **comp** é incrementada.

Algoritmo 7. Componentes conexos

```
def busca_largura_conexos(G, s, comp):
    Q = [s]
    R = [s]
    desc[s] = comp
    while len(Q) != 0:
        u = Q.pop(0)
        for v, i in G[u]:
            if desc[v] == 0:
                Q.append(v)
                R.append(v)
                desc[v] = comp
    return R


def componentes_conexas(G):
    comp = 0
    for v in range(len(G)):
        if desc[v] == 0:
            comp += 1
            busca_largura_conexos(G, v, comp)
```

Para exibir as informações das componentes conexas foi utilizada a função abaixo, chamando a função **componentes-conexas** para descobrir as componentes conexas. Em **desc** cada posição com um número diferente vai corresponder a uma componente conexa, essas componentes são armazenadas em **conexos** nas linhas (5-7), logo depois a quantidade de componentes conexas do grafo e a quantidade de vértices que cada uma delas possui são exibidas, também é mostrado a quantidade de vértices da maior e da menor componente conexa ao final da função. No caso de uma matriz, ela é transformada em uma lista em uma outra função e logo depois esta é chamada.

Algoritmo 8. Informações componentes conexas

```
def informacoesGrafosConexos(grafo):
    componentes_conexas(grafo)

    conexos = []
    for i in desc:
        if i not in conexos:
            conexos.append(i)
    conexos.sort()

    print("Componentes conexas:", len(conexos))
```

```
11      input("\nAperte enter para exibir a quantidade de vertices conexos
            em cada componente !\n")
12
13      for i in conexos:
14          print("-", desc.count(i), "vertices")
15
16      frequencia = Counter(desc).most_common()
17      print("\nA maior componente conexa possui", frequencia[0][1], "
            vertice(s) !\n")
18      print("A menor componente conexa possui", frequencia[-1][1], "
            vertice(s) !\n")
```

3. Experimentos e Resultados

Nesta seção os resultados obtidos através dos experimentos serão exibidos, analisados e comentados. Os testes foram feitos com base nas perguntas realizadas no enunciado do trabalho por um notebook com processador Intel Core i5-8265U com 16GB de memória RAM e quatro núcleos operando a 1.60GHz cada.

3.1. Grafo de Colaborações em Pesquisa

Para este grafo não foi possível realizar nenhum tipo de experimento utilizando a matriz de adjacências, pois para a sua representação seriam necessários 71998^2 posições de memória e nem mesmo um notebook com 16GB de memória RAM é capaz de carregá-la.

Já na lista de adjacências foi possível realizar todos os experimentos propostos, após ser reiniciado e somente com o VS Code aberto o notebook registrava 4.5GB de memória utilizada como mostra a Figura 1 e quando a representação do grafo por lista de adjacências foi realizada ele marcou 4.6GB de memória utilizada, como mostra a Figura 2. Chegando a conclusão de que a representação por lista de adjacências gasta por volta de 100MB de memória.

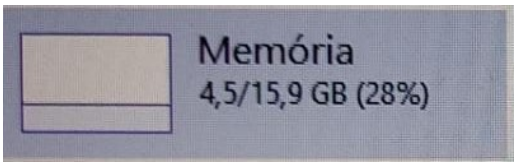

Figura 1

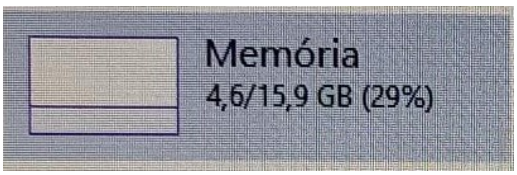

Figura 2

Utilizando o vértice 1 como ponto de partida as buscas em largura e em profundiade ficaram na casa do 1 segundo quando se compara o tempo de execução. Seria necessário um cronômetro para chegar na casa dos milésimos e medir o tempo com mais exatidão.

O maior grau encontrado no grafo foi do vértice 72 com grau 3691 e o menor grau é do vértice 31 que possui grau 0. Pela quantidade de vértices existentes no grafo era esperado um maior grau bem superior, sendo assim, é possível esperar que este grafo possua várias componentes conexas. A distribuição empírica do grau dos vértices pode ser observada na Figura 3.

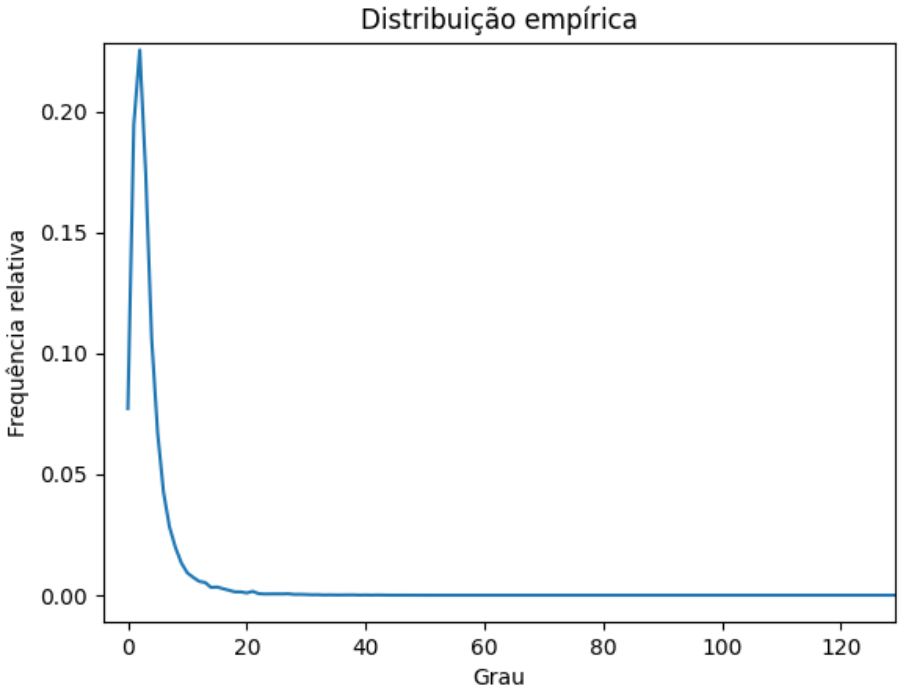

Figura 3

Após realizar os teste de componentes conexas foi possível ver que o grafo possui várias, como era de se esperar, 14384 para ser mais exato. A maior componente conexa possui 33533 vértices e a menor possui apenas 1 vértice.

3.2. Grafo de Conexões da Web

No grafo de conexões da web foi possível realizar os testes representando ele tanto por lista de adjacência quanto por matriz de adjacência, isso foi possível por se tratar de um grafo menor, sendo possível armazenar a matriz de adjacência sem dificuldades e em ambos os casos de representação os resultados obtidos foram os mesmos.

O maior grau encontrado no grafo foi 2159 do vértice 1 e o menor é de grau 1 do vértice 0, apesar de ser um grau bem inferior ao número de vértices (32385), o grau deste grafo já é consideravelmente maior que o do grafo anterior que possuia apenas um maior grau de 72 para um grafo de 71998 vértices. A distribuição empírica completa pode ser vista na Figura 5, mas como a frequência relativa do grau 2159 é bem próxima de zero, na Figura 4 é possível ver a distribuição empírica com mais detalhes.

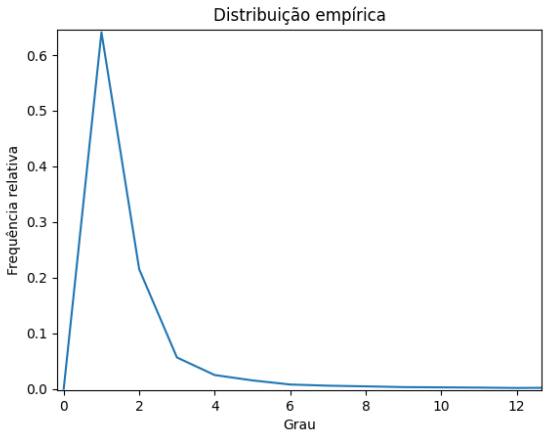

Figura 4

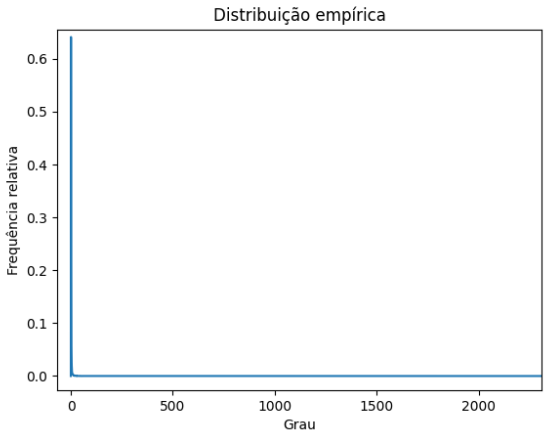

Figura 5

Apesar de possuir um maior grau não muito alto o grafo possui apenas uma componente conexa com todos os 32385 vértices, mas como se trata de um grafo sobre as conexões web já era de se esperar que elas estivessem ligadas entre si.

Através da busca em largura no grafo foi possível ver que independente do vértice em que a busca é inicada o grafo mantém um nível padrão entre 6 e 9 como mostra a Tabela 1, onde também é exibido a resultado de algumas outras buscas que foram realizadas.

Tabela 1. Busca em largura

Vértice origem	Nível	Vértice origem	Nível
0	8	10000	8
1	6	20000	7
2	7	16192	8
10	6	30000	7
100	7	32384	9

O diâmetro da internet também foi determinado com o uso da busca em largura, este diâmetro da internet nada mais é do que a maior distância entre qualquer par de vértices do grafo, ou seja, o maior nível encontrado no grafo, que no caso é 9 quando a busca é iniciado pelo vértice 32384.

4. Conclusão

Chegando ao fim do trabalho conclui-se que os algoritmos de busca são muito úteis e realmente podem ajudar na resolução de problemas reais. Neste artigo em específico eles foram responsáveis por facilitar a análise de grafos enormes e sem eles esta análise levaria muito mais tempo para acontecer.

Através do artigo foi possível observar que para a modelagem de grandes problemas da vida real em grafos de forma computacional é melhor optar pela lista de adjacências, pois é uma forma simples, rápida, que não consome tanta memória mesmo em casos de grafos gigantes e não é necessário um super computador para utilizá-la. Além disso o estudo dos tipos de busca e de como elas funcionam é importante, pois trata-se de uma forma de resolução de problemas atual e bastante utilizado.

Referências

FONSECA, George Henrique Godim da. Prof. George Fonseca, 2021. A06 Busca em Largura e em Profundidade. Disponível em: http://professor.ufop.br/george/classes/csi466. Acesso em: 16 de julho de 2021.

COTA, Lucas Monteiro Martino. GitHub, 2021. Busca em Largura e em Profundidade. Disponível em: https://github.com/lucasmmcota/graphs. Acesso em: 16 de julho de 2021.